APPENDIX A

APPENDIX A

Card Class File Format For Preferred Embodiment

Introduction

The card class file is a compressed form of the original class file(s). The card class file contains onl semantic information required to interpret Java programs from the original class files. The indirect references in the original class file are replaced with direct references resulting in a compact repres The card class file format is based on the following principles:

1. *Stay close to the standard class file format*: The card class file format should remain as close standard class file format as possible. The Java byte codes in the class file remain unaltered. N altering the byte codes ensures that the structural and static constraints on them remain verifiab

2. *Ease of implementation*: The card class file format should be simple enough to appeal to Java Machine implementers. It must allow for different yet behaviorally equivalent implementations

3. *Feasibility*: The card class file format must be compact in order to accommodate smart card technology. It must meet the constraints of today's technology while not losing sight of tomorro innovations.

This document is based on Chapter 4, "The class file format", in the book titled "The Java™ Virtua Machine Specification"[1], henceforth referred to as the Red book. Since the document is based on standard class file format described in the Red book, we only present information that is different. 1 book serves as the final authority for any clarification.

The primary changes from the standard class file format are:

*   The constant pool is optimized to contain only 16-bit identifiers and, where possible, indirectio replaced by a direct reference.
*   Attributes in the original class file are eliminated or regrouped. ·

The Java Card class File Format

This section describes the Java Card class file format. Each card class file contains one or many Jav where a type may be a class or an interface.

A card class file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities are cons by reading in two, four, and eight consecutive 8-bit bytes, respectively. Multi-byte data items are al stored in big-endian order, where the high bytes come first. In Java, this format is supported by inte java.io.DataInput and java.io.DataOutput and classes such as java.io.DataInputStream and java.io.DataOutputStream.

We define and use the same set of data types representing Java class file data: The types u1, u2, and represent an unsigned one-, two-, or four-byte quantity, respectively. In Java, these types may be re methods such as readUnsignedByte, readUnsignedShort, and readInt of the interface java.io.DataIn The card class file format is presented using pseudo-structures written in a C-like structure notation. avoid confusion with the fields of Java Card Virtual Machine classes and class instances, the conten structures describing the card class file format are referred to as items. Unlike the fields of a C struc successive items are stored in the card class file sequentially, without padding or alignment.

Variable-sized tables, consisting of variable-sized items, are used in several class file structures. Alt we will use C-like array syntax to refer to table items, the fact that tables are streams of varying-size structures means that it is not possible to directly translate a table index into a byte offset into the tal Where we refer to a data structure as an array, it is literally an array.

In order to distinguish between the card class file structure and the standard class file structure, we a capitalization; for example, we rename field_info in the original class file to FieldInfo in the card cl

Card Class File

A card class file contains a single CardClassFile structure:

```
CardClassFile {
    u1 major_version;
    u1 minor_version;
    u2 name_index;
    u2 const_size;
    u2 max_class:
    CpInfo constant_pool[const_size];
    ClassInfo class[max_class];
}
```

The items in the CardClassFile structure are as follows:

minor_version, major_version

The values of the minor_version and major_version items are the minor and major version numbers of off-card Java Card Virtual Machine that produced this card class file. An implementation of the Java Virtual Machine normally supports card class files having a given major version number and minor v numbers 0 through some particular minor_version.

Only the Java Card Forum may define the meaning of card class file version numbers.

name_index

The value of the name_index item must represent a valid Java class name. The Java class name repre by name_index must be exactly the same Java class name that corresponds to the main application th run in the card. A card class file contains several classes or interfaces that constitute the application t runs in the card. Since Java allows each class to contain a main method there must be a way to distin; the class file containing the main method which corresponds to the card application.

const_size

The value of const_size gives the number of entries in the card class file constant pool. A constant_p index is considered valid if it is greater than or equal to zero and less than const_size.

max_class

This value refers to the number of classes present in the card class file. Since the name resolution and linking in the Java Card are done by the off-card Java Virtual Machine all the class files or classes re for an application are placed together in one card class file.

constant_pool[]

The constant_pool is a table of variable-length structures (0) representing various string constants, cl names, field names, and other constants that are referred to within the CardClassFile structure and its substructures.

The first entry in the card class file is constant_pool[0].

Each of the constant_pool table entries at indices 0 through const_size is a variable-length structure (

class[]

The class is a table of max_class classes that constitute the application loaded onto the card.

Constant Pool

All constant_pool table entries have the following general format:

```
CpInfo {
    u1 tag;
    u1 info[];
}
```

Each item in the constant_pool table must begin with a 1-byte tag indicating the kind of cp_info entr contents of the info array varies with the value of tag. The valid tags and their values are the same as specified in the Red book.

Each tag byte must be followed by two or more bytes giving information about the specific constant. format of the additional information varies with the tag value. Currently the only tags that need to be included are CONSTANT_Class, CONSTANT_FieldRef, CONSTANT_MethodRef and CONSTANT_InterfaceRef. Support for other tags be added as they are included in the specification.

CONSTANT_Class

The CONSTANT_Class_info structure is used to represent a class or an interface:
```
CONSTANT_ClassInfo {
    u1 tag;
    u2 name_index;
}
```
The items of the CONSTANT_Class_info structure are the following:

tag

The tag item has the value CONSTANT_Class (7).

name_index

The value of the name_index item must represent a valid Java class name. The Java class name represe
by name_index must be exactly the same Java class name that is described by the corresponding
CONSTANT_Class entry in the constant_pool of the original class file.

CONSTANT_Fieldref, CONSTANT_Methodref, and CONSTANT_InterfaceMethodref

Fields, methods, and interface methods are represented by similar structures:
```
CONSTANT_FieldrefInfo {
    u1 tag;
    u2 class_index;
    u2 name_sig_index;
}
CONSTANT_MethodrefInfo {
    u1 tag;
    u2 class_index;
    u2 name_sig_index;
}
CONSTANT_InterfaceMethodrefInfo {
    u1 tag;
    u2 class_index;
    u2 name_sig_index;
}
```
The items of these structures are as follows:

tag

The tag item of a CONSTANT_FieldrefInfo structure has the value CONSTANT_Fieldref (9).

The tag item of a CONSTANT_MethodrefInfo structure has the value CONSTANT_Methodref (10).

The tag item of a CONSTANT_InterfaceMethodrefInfo structure has the value
CONSTANT_InterfaceMethodref (11).

classs_index

The value of the class_index item must represent a valid Java class or interface name. The name repres
by class_index must be exactly the same name that is described by the corresponding
CONSTANT_Class_info entry in the constant_pool of the original class file.

name_sig_index

The value of the name_sig_index item must represent a valid Java name and type. The name and type
represented by name_sig_index must be exactly the same name and type described by the
CONSTANT_NameAndType_info entry in the constant_pool structure of the original class file.

Class

Each class is described by a fixed-length ClassInfo structure. The format of this structure is:
```
ClassInfo {
    u2 name_index;
    u1 max_field;
    u1 max_sfield;
    u1 max_method;
    u1 max_interface;
    u2 superclass;
    u2 access_flags;
```

```
            FieldInfo field[max_field+max_sfield];
            InterfaceInfo interface[max_interface];
            MethodInfo method[max_method];
    }
```

The items of the ClassInfo structure are as follows:

name_index

The value of the name_index item must represent a valid Java class name. The Java class name repre by name_index must be exactly the same Java class name that is described in the corresponding Clas structure of the original class file.

max_field

The value of the max_field item gives the number of FieldInfo (0) structures in the field table that re the instance variables, declared by this class or interface type. This value refers to the number of non the fields in the card class file. If the class represents an interface the value of max_field is 0.

max_sfield

The value of the max_sfield item gives the number of FieldInfo structures in the field table that repre the class variables, declared by this class or interface type. This value refers to the number of static t fields in the card class file.

max_method

The value of the max_method item gives the number of MethodInfo (0) structures in the method tabl

max_interface

The value of the max_interface item gives the number of direct superinterfaces of this class or interfa type.

superclass

For a class, the value of the superclass item must represent a valid Java class name. The Java class na represented by superclass must be exactly the same Java class name that is described in the correspoi ClassFile structure of the original class file. Neither the superclass nor any of its superclasses may be class.

If the value of superclass is 0[1], then this class must represent the class java.lang.Object, the only clas interface without a superclass.

For an interface, the value of superclass must always represent the Java class java.lang.Object.

access_flags

The value of the access_flags item is a mask of modifiers used with class and interface declarations. access_flags modifiers and their values are the same as the access_flags modifiers in the correspondi ClassFile structure of the original class file.

field[]

Each value in the field table must be a fixed-length FieldInfo (0) structure giving a complete descript a field in the class or interface type. The field table includes only those fields that are declared by thi or interface. It does not include items representing fields that are inherited from superclasses or superinterfaces.

interface[]

Each value in the interface array must represent a valid interface name. The interface name represent each entry must be exactly the same interface name that is described in the corresponding interface ai the original class file.

method[]

Each value in the method table must be a variable-length MethodInfo (0) structure giving a complete description of and Java Virtual Machine code for a method in the class or interface.

The MethodInfo structures represent all methods, both instance methods and, for classes, class (static methods, declared by this class or interface type. The method table only includes those methods that. explicitly declared by this class. Interfaces have only the single method <clinit>, the interface initiali; method. The methods table does not include items representing methods that are inherited from super or superinterfaces.

---

[1] Or a standard yet fixed value.

Fields

Each field is described by a fixed-length field_info structure. The format of this structure is

```
FieldInfo {
    u2 name_index;
    u2 signature_index;
    u2 access_flags;
}
```

The items of the FieldInfo structure are as follows:

name_index

The value of the name_index item must represent a valid Java field name. The Java field name repre by name_index must be exactly the same Java field name that is described in the corresponding field structure of the original class file.

signature_index

The value of the signature_index item must represent a valid Java field descriptor. The Java field des represented by signature index must be exactly the same Java field descriptor that is described in the corresponding field_info structure of the original class file.

access_flags

The value of the access_flags item is a mask of modifiers used to describe access permission to and properties of a field. The access_flags modifiers and their values are the same as the access_flags mc in the corresponding field_info structure of the original class file.

Methods

Each method is described by a variable-length MethodInfo structure. The MethodInfo structure is a variable-length structure that contains the Java Virtual Machine instructions and auxiliary informatio single Java method, instance initialization method, or class or interface initialization method. The str has the following format:

```
MethodInfo {
    u2 name_index;
    u2 signature_index;
    u1 max_local;
    u1 max_arg;
    u1 max_stack;
    u1 access_flags;
    u2 code_length;
    u2 exception_length;
    u1 code[code_length];
    {    u2 start_pc;
         u2 end_pc;
         u2 handler_pc;
         u2 catch_type;
    } einfo[exception_length];
}
```

The items of the MethodInfo structure are as follows:

name_index

The value of the name_index item must represent either one of the special internal method names, eit <init> or <clinit>, or a valid Java method name. The Java method name represented by name_index i exactly the same Java method name that is described in the corresponding method_info structure of t original class file.

signature_index

The value of the signature_index item must represent a valid Java method descriptor. The Java methc descriptor represented by signature_index must be exactly the same Java method descriptor that is de in the corresponding method_info structure of the original class file.

max_local

The value of the max_locals item gives the number of local variables used by this method, excluding parameters passed to the method on invocation. The index of the first local variable is 0. The greates variable index for a one-word value is max_locals-1.

max_arg

The value of the max_arg item gives the maximum number of arguments to this method.

max_stack

The value of the max_stack item gives the maximum number of words on the operand stack at any p· during execution of this method.

access_flags

The value of the access_flags item is a mask of modifiers used to describe access permission to and properties of a method or instance initialization method. . The access_flags modifiers and their value the same as the access_flags modifiers in the corresponding method_info structure of the original cla

code_length

The value of the code_length item gives the number of bytes in the code array for this method. The ν code_length must be greater than zero; the code array must not be empty.

exception_length

The value of the exception_length item gives the number of entries in the exception_info table.

code[]

The code array gives the actual bytes of Java Virtual Machine code that implement the method. Whe code array is read into memory on a byte addressable machine, if the first byte of the array is aligned byte boundary, the tableswitch and lookupswitch 32-bit offsets will be 4-byte aligned; refer to the descriptions of those instructions for more information on the consequences of code array alignment The detailed constraints on the contents of the code array are extensive and are the same as describe Java Virtual Machine Specification.

einfo[]

Each entry in the einfo array describes one exception handler in the code array. Each einfo entry con the following items:

start_pc, end_pc

The values of the two items start_pc and end_pc indicate the ranges in the code array at which the e handler is active.

The value of start_pc must be a valid index into the code array of the opcode of an instruction. The ν end_pc either must be a valid index into the code array of the opcode of an instruction, or must be e code_length, the length of the code array. The value of start_pc must be less than the value of end_p The start_pc is inclusive and end_pc is exclusive; that is, the exception handler must be active while program counter is within the interval [start_pc, end_pc].

handler_pc

The value of the handler_pc item indicates the start of the exception handler. The value of the item r a valid index into the code array, must be the index of the opcode of an instruction, and must be less the value of the code_length item.

catch_type

If the value of the catch_type item is nonzero, it must represent a valid Java class type. The Java cla: represented by catch_type must be exactly the same as the Java class type that is described by the catch_type in the corresponding method_info structure of the original class file. This class must be tl Throwable or one of its subclasses. The exception handler will be called only if the thrown exceptio: instance of the given class or one of its subclasses.

If the value of the catch_type item is zero, this exception handler is called for all exceptions. This is implement finally.

Attributes

Attributes used in the original class file are either eliminated or regrouped for compaction.
The predefined attributes SourceFile, ConstantValue, Exceptions, LineNumberTable, and Local-VariableTable may be eliminated without sacrificing any information required for Java byte code interpretation.

A-6

The predefined attribute Code which contains all the byte codes for a particular method are moved in t
corresponding MethodInfo structure.

Constraints on Java Card Virtual Machine Code

The Java Card Virtual Machine code for a method, instance initialization method. or class or interface
initialization method is stored in the array code of the MethodInfo structure of a card class file. Both tl
static and the structural constraints on this code array are the same as those described in the Red book.
Limitations of the Java Card Virtual Machine and Java Card class File Format
The following limitations in the Java Card Virtual Machine are imposed by this version of the Java Ca
Virtual Machine specification:

- The per-card class file constant pool is limited to 65535 entries by the 16-bit const_size field of tl
  CardClassFile structure (0). This acts as an internal limit on the total complexity of a single card (
  file. This count also includes the entries corresponding to the constant pool of the class hierarchy
  available to the application in the card.[2]
- The amount of code per method is limited to 65535 bytes by the sizes of the indices in the Metho
  structure.
- The number of local variables in a method is limited to 255 by the size of the max_local item of t
  MethodInfo structure (0).
- The number of fields of a class is limited to 510 by the size of the max_field and the max_sfield i
  of the ClassInfo structure (0).
- The number of methods of a class is limited to 255 by the size of the max_method item of the Cl
  structure (0).
- The size of an operand stack is limited to 255 words by the max_stack field of the MethodInfo st
  (0).

Bibliography

[1] Tim Lindholm and Frank Yellin, The Java Virtual Machine Specification, Addison-Wesley, 1996

---

[2] A single card class file constant pool has 65535-Δ entries available, where Δ corresponds to the nu
entries in the constant pool of the class hierarchies accessible to the application.

APPENDIX B

APPENDIX B

String To ID Input And Output

For the correct operation of Card JVM it is very important that the declared and generated IDs are managed. This management is controlled by the definitions in the string to ID input file **String-ID** This textual file, the basis for which is shown below, declares which areas of the namespace can be what purposes. One possible arrangement of this map may reserve some IDs for internal use by the JVM interpreter, and the rest is allocated to Card JVM applications.

```
#
# String-ID INMap file.
#
#       4000 - 7FFF    Available for application use.
#       F000 - FFFE    Reserved for Card JVM's internal use.
#
constantBase        F000        # The area from F000 to FFFF is reserved for
                                # Card JVM's internal use.
                                #
MainApplication                 # F000 - Name of the startup class
                                # (changes for each application)
main()V                         # F001 - Name of the startup method
                                # (may change for each application)
java/lang/Object                # F002
java/lang/String                # F003
<init>()V                       # F004
<clinit>()V                     # F005
[L                              # F006
[I                              # F007
[C                              # F008
[B                              # F009
[S                              # F000A
#
constantBase        FFF0        # This area is reserved for simple return types
L                               # FFF0
V                               # FFF1
I                               # FFF2
S                               # FFF3
C                               # FFF4
B                               # FFF5
Z                               # FFF6
#
constantBase        4000        # From here on this space is application depend
```

Essentially, all applications which are to be loaded into a smart card are allocated their own IDs with 0x4000 to 0x7FFF. This space is free for each application since no loaded application is permitted to other applications.

Care must be taken on managing the IDs for preloaded class libraries. The management of these IDs helped by the (optional) generation of the string to ID output file **String-ID OUTMap** file. This map **String-ID INMap** augmented with the new String-ID bindings. These bindings may be produced wh Card Class File Converter application terminates. The **String-ID OUTMap** is generated for support libraries and OS interfaces loaded on the card. This map may be used as the **String-ID INMap** for s card applications using the support libraries and OS interfaces loaded on the card. When building ne applications this file can generally be discarded.

As an example consider the following Java program. HelloSmartCard.java. When compiled it generat class file HelloSmartCard.class. This class file has embedded in it strings that represent the class nam methods and type information. On the basis of the **String-ID INMap** described above Card Class Fil Converter generates a card class file that replaces the strings present in the class file with IDs allocate Card Class File Converter. Table 1 lists the strings found in the constant pool of HelloSmartCard.clas their respective Card Class File Converter assigned IDs. Note that some strings (like "java/lang/Object") have a pre-assigned value (F002) and some strings (like "()V") get a ne value (4004).

```
public class HelloSmartCard {
  public byte aVariable;

  public static void main() {
    HelloSmartCard h = new HelloSmartCard();
    h.aVariable = (byte)13;
  }
}
```

Program : HelloSmartCard.java

Offset (in Constant Pool)	String	ID	Mapped New/ Mapped/Old
00000A	"Code"	4000	New
000011	"SourceFile"	4001	New
00001E	"ConstantValue"	4002	New
00002E	"Exceptions"	4003	New
00003B	"HelloSmartCard"	F000	Old
00004C	"java/lang/Object"	F002	Old
000062	"<init>"	F004	Old
00006E	"()V"	4004	New
000074	"aVariable"	4005	New
00008A	"B"	FFF5	Old
00008E	"HelloSmartCard.java"	4006	New
0000B3	"main"	F001	Old

Relevant entries of String-ID OUTMap
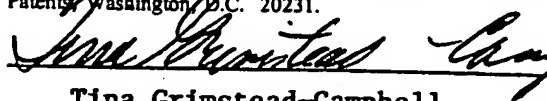
B-2

APPENDIX C

APPENDIX C

Byte codes supported by the Card JVM in the preferred embodiment

AALOAD	AASTORE	ACONST_NULL
ALOAD	ALOAD_0	ALOAD_1
ALOAD_2	ALOAD_3	ARETURN
ARRAYLENGTH	ASTORE	ASTORE_0
ASTORE_1	ASTORE_2	ASTORE_3
ATHROW	BALOAD	BASTORE
CHECKCAST	DUP	DUP2
DUP2_X1	DUP2_X2	DUP_X1
DUP_X2	GETFIELD	GETSTATIC
GOTO	IADD	IALOAD
IAND	IASTORE	ICONST_0
ICONST_1	ICONST_2	ICONST_3
ICONST_4	ICONST_5	ICONST_M1
IDIV	IFEQ	IFGE
IFGT	IFLE	IFLT
IFNE	IFNONNULL	IFNULL
IF_ACMPEQ	IF_ACMPNE	IF_ICMPEQ
IF_ICMPGE	IF_ICMPGT	IF_ICMPLE
IF_ICMPLT	IF_ICMPNE	IINC
ILOAD	ILOAD_0	ILOAD_1
ILOAD_2	ILOAD_3	IMUL
INEG	INSTANCEOF	INT2BYTE
INT2CHAR	INT2SHORT	INVOKEINTERF/
INVOKENONVIRTUAL	INVOKESTATIC	INVOKEVIRTUA
IOR	IREM	IRETURN
ISHL	ISHR	ISTORE
ISTORE_0	ISTORE_1	ISTORE_2
ISTORE_3	ISUB	IUSHR
IXOR	JSR	LDC1
LDC2	LOOKUPSWITCH	NEW
NEWARRAY	NOP	POP
POP2	PUTFIELD	PUTSTATIC
RET	RETURN	SALOAD
SASTORE	SIPUSH	SWAP
TABLESWITCH	BIPUSH	

Standard Java byte codes numbers for the byte codes supported in the preferred embodiment

```
package util;

/*
 * List of actual Java Bytecodes handled by this JVM
 * ref. Lindohlm and Yellin.
 *
 * Copyright (c) 1996 Schlumberger Austin Products Center,
 *                     Schlumberger, Austin, Texas, USA.
 */

public interface BytecodeDefn (
    public static final byte j_NOP = (byte)0;
    public static final byte ACONST_NULL = (byte)1;
    public static final byte ICONST_M1 = (byte)2;
    public static final byte ICONST_0 = (byte)3;
    public static final byte ICONST_1 = (byte)4;
    public static final byte ICONST_2 = (byte)5;
    public static final byte ICONST_3 = (byte)6;
    public static final byte ICONST_4 = (byte)7;
    public static final byte ICONST_5 = (byte)8;
    public static final byte BIPUSH = (byte)16;
    public static final byte SIPUSH = (byte)17;
    public static final byte LDC1 = (byte)18;
    public static final byte LDC2 = (byte)19;
    public static final byte ILOAD = (byte)21;
    public static final byte ALOAD = (byte)25;
    public static final byte ILOAD_0 = (byte)26;
    public static final byte ILOAD_1 = (byte)27;
    public static final byte ILOAD_2 = (byte)28;
    public static final byte ILOAD_3 = (byte)29;
    public static final byte ALOAD_0 = (byte)42;
    public static final byte ALOAD_1 = (byte)43;
    public static final byte ALOAD_2 = (byte)44;
    public static final byte ALOAD_3 = (byte)45;
    public static final byte IALOAD = (byte)46;
    public static final byte AALOAD = (byte)50;
    public static final byte BALOAD = (byte)51;
    public static final byte CALOAD = (byte)52;
    public static final byte ISTORE = (byte)54;
    public static final byte ASTORE = (byte)58;
    public static final byte ISTORE_0 = (byte)59;
    public static final byte ISTORE_1 = (byte)60;
    public static final byte ISTORE_2 = (byte)61;
    public static final byte ISTORE_3 = (byte)62;
    public static final byte ASTORE_0 = (byte)75;
    public static final byte ASTORE_1 = (byte)76;
    public static final byte ASTORE_2 = (byte)77;
    public static final byte ASTORE_3 = (byte)78;
    public static final byte IASTORE = (byte)79;
    public static final byte AASTORE = (byte)83;
    public static final byte BASTORE = (byte)84;
    public static final byte CASTORE = (byte)85;
    public static final byte POP = (byte)87;
    public static final byte POP2 = (byte)88;
    public static final byte DUP = (byte)89;
    public static final byte DUP_X1 = (byte)90;
    public static final byte DUP_X2 = (byte)91;
    public static final byte DUP2 = (byte)92;
    public static final byte DUP2_X1 = (byte)93;
    public static final byte DUP2_X2 = (byte)94;
    public static final byte SWAP = (byte)95;
    public static final byte IADD = (byte)96;
    public static final byte ISUB = (byte)100;
    public static final byte IMUL = (byte)104;
    public static final byte IDIV = (byte)108;
    public static final byte IREM = (byte)112;
```

```java
    public static final byte INEG = (byte)116;
    public static final byte ISHL = (byte)120;
    public static final byte ISHR = (byte)122;
    public static final byte IUSHR = (byte)124;
    public static final byte IAND = (byte)126;
    public static final byte IOR = (byte)128;
    public static final byte IXOR = (byte)130;
    public static final byte IINC = (byte)132;
    public static final byte INT2BYTE = (byte)145;
    public static final byte INT2CHAR = (byte)146;
    public static final byte INT2SHORT = (byte)147;
    public static final byte IFEQ = (byte)153;
    public static final byte IFNE = (byte)154;
    public static final byte IFLT = (byte)155;
    public static final byte IFGE = (byte)156;
    public static final byte IFGT = (byte)157;
    public static final byte IFLE = (byte)158;
    public static final byte IF_ICMPEQ = (byte)159;
    public static final byte IF_ICMPNE = (byte)160;
    public static final byte IF_ICMPLT = (byte)161;
    public static final byte IF_ICMPGE = (byte)162;
    public static final byte IF_ICMPGT = (byte)163;
    public static final byte IF_ICMPLE = (byte)164;
    public static final byte IF_ACMPEQ = (byte)165;
    public static final byte IF_ACMPNE = (byte)166;
    public static final byte GOTO = (byte)167;
    public static final byte j_JSR = (byte)168;
    public static final byte RET = (byte)169;
    public static final byte TABLESWITCH = (byte)170;
    public static final byte LOOKUPSWITCH = (byte)171;
    public static final byte IRETURN = (byte)172;
    public static final byte ARETURN = (byte)176;
    public static final byte RETURN = (byte)177;
    public static final byte GETSTATIC = (byte)178;
    public static final byte PUTSTATIC = (byte)179;
    public static final byte GETFIELD = (byte)180;
    public static final byte PUTFIELD = (byte)181;
    public static final byte INVOKEVIRTUAL = (byte)182;
    public static final byte INVOKENONVIRTUAL = (byte)183;
    public static final byte INVOKESTATIC = (byte)184;
    public static final byte INVOKEINTERFACE = (byte)185;
    public static final byte NEW = (byte)187;
    public static final byte NEWARRAY = (byte)188;
    public static final byte ARRAYLENGTH = (byte)190;
    public static final byte ATHROW = (byte)191;
    public static final byte CHECKCAST = (byte)192;
    public static final byte INSTANCEOF = (byte)193;
    public static final byte IFNULL = (byte)198;
    public static final byte IFNONNULL = (byte)199;
}
```
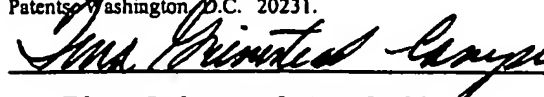
C-3

APPENDIX D

APPENDIX D

Card Class File Converter byte code conversion process

```
/*
 * Reprocess code block.
 */
static
void
reprocessMethod(iMethod* imeth)
{
    int pc;
    int npc;
    int align;
    bytecode* code;
    int codelen;
    int i;
    int opad;
    int npad;
    int apc;
    int high;
    int low;

/* codeinfo is a table that keeps track of the valid Java bytecodes and their
 * corresponding translation
 */
    code = imeth->external->code;
    codelen = imeth->external->code_length;

    jumpPos = 0;
    align = 0;

    /* Scan for unsupported opcodes */
    for (pc = 0; pc < codelen; pc = npc) {
      if (codeinfo[code[pc]].valid == 0) {
        error("Unsupported opcode %d", code[pc]);
      }
      npc = nextPC(pc, code);
    }

    /* Scan for jump instructions an insert into jump table */
    for (pc = 0; pc < codelen; pc = npc) {
      npc = nextPC(pc, code);

      if (codeinfo[code[pc]].valid == 3) {
        insertJump(pc+1, pc, (int16)((code[pc+1] << 8)|code[pc+2]));
      }
      else if (codeinfo[code[pc]].valid == 4) {
        apc = pc & -4;
        low = (code[apc+8] << 24) | (code[apc+9] << 16)
              | (code[apc+10] << 8) | code[apc+11];
        high = (code[apc+12] << 24) | (code[apc+13] << 16)
              | (code[apc+14] << 8) | code[apc+15];
        for (i = 0; i < high-low+1; i++) {
          insertJump(apc+(i*4)+18, pc,
                    (int16)((code[apc+(i*4)+18] << 8) | code[apc+(i*4)+19]));
        }
        insertJump(apc+6, pc, (int16)((code[apc+6] << 8) | code[apc+7]));
      }
      else if (codeinfo[code[pc]].valid == 5) {
        apc = pc & -4;
        low = (code[apc+8] << 24) | (code[apc+9] << 16)
              | (code[apc+10] << 8) | code[apc+11];
        for (i = 0; i < low; i++) {
          insertJump(apc+(i*8)+18, pc,
                    (int16)((code[apc+(i*8)+18] << 8) | code[apc+(i*8)+19]));
        }
        insertJump(apc+6, pc, (int16)((code[apc+6] << 8) | code[apc+7]));
      }
    }
```

A

B

```c
#ifdef TRANSLATE_BYTECODE
    /* Translate specific opcodes to general ones */
    for (pc = 0; pc < codelen; pc = npc) {
        /* This is a translation code */
        if (codeinfo[code[pc]].valid == 2) {
            switch (code[pc]) {
            case ILOAD_0:
            case ILOAD_1:
            case ILOAD_2:
            case ILOAD_3:
                insertSpace(code, &codelen, pc, 1);
                align += 1;
                code[pc+1] = code[pc] - ILOAD_0;
                code[pc+0] = ILOAD;
                break;

            case ALOAD_0:
            case ALOAD_1:
            case ALOAD_2:
            case ALOAD_3:
                insertSpace(code, &codelen, pc, 1);
                align += 1;
                code[pc+1] = code[pc] - ALOAD_0;
                code[pc+0] = ALOAD;
                break;

            case ISTORE_0:
            case ISTORE_1:
            case ISTORE_2:
            case ISTORE_3:
                insertSpace(code, &codelen, pc, 1);
                align += 1;
                code[pc+1] = code[pc] - ISTORE_0;
                code[pc+0] = ISTORE;
                break;

            case ASTORE_0:
            case ASTORE_1:
            case ASTORE_2:
            case ASTORE_3:
                insertSpace(code, &codelen, pc, 1);
                align += 1;
                code[pc+1] = code[pc] - ASTORE_0;
                code[pc+0] = ASTORE;
                break;

            case ICONST_M1:
                insertSpace(code, &codelen, pc, 2);
                align += 2;
                code[pc+2] = 255;
                code[pc+1] = 255;
                code[pc+0] = SIPUSH;
                break;

            case ICONST_0:
            case ICONST_1:
            case ICONST_2:
            case ICONST_3:
            case ICONST_4:
            case ICONST_5:
                insertSpace(code, &codelen, pc, 2);
                align += 2;
                code[pc+2] = code[pc] - ICONST_0;
                code[pc+1] = 0;
                code[pc+0] = SIPUSH;
                break;

            case LDC1:
                insertSpace(code, &codelen, pc, 1);
                align += 1;
                code[pc+1] = 0;
                code[pc+0] = LDC2;
                break;
```

```
                    case BIPUSH:
                      insertSpace(code, &codelen, pc, 1);
                      align += 1;
                      if ((int8)code[pc+2] >= 0) {
                        code[pc+1] = 0;
                      }
                      else {
                        code[pc+1] = 255;
                      }
                      code[pc+0] = SIPUSH;
                      break;

                    case INT2SHORT:
                      removeSpace(code, &codelen, pc, 1);
                      align -= 1;
                      npc = pc;
                      continue;

                    }
                }
                else if (codeinfo[code[pc]].valid == 4 || codeinfo[code[pc]].valid == 5) {
                    /* Switches are aligned to 4 byte boundaries.  Since we are inserting and
                     * removing bytecodes, this may change the alignment of switch instructions.
                     * Therefore, we must readjust the padding in switches to compensate.
                     */
                    opad = (4 - (((pc+1) - align) % 4)) % 4; /* Current switch padding */
                    npad = (4 - ((pc+1) % 4)) % 4;          /* New switch padding */
                    if (npad > opad) {
                      insertSpace(code, &codelen, pc+1, npad - opad);
                      align += (npad - opad);
                    }
                    else if (npad < opad) {
                      removeSpace(code, &codelen, pc+1, opad - npad);
                      align -= (opad - npad);
                    }
                }

                npc = nextPC(pc, code);
            }
        #endif

            /* Relink constants */
            for (pc = 0; pc < codelen; pc = npc) {
                npc = nextPC(pc, code);
                i = (uint16)((code[pc+1] << 8) + code[pc+2]);

                switch (code[pc]) {
                case LDC2:
                    /* 'i' == general index */
                    switch (cItem(i).type) {
                    case CONSTANT_Integer:
                      i = cItem(i).v.tint;
                      code[pc] = SIPUSH;
                      break;

                    case CONSTANT_String:
                      i = buildStringIndex(i);
                      break;

                    default:
                      error("Unsupported loading of constant type");
                      break;
                    }
                    break;

                case NEW:
                case INSTANCEOF:
                case CHECKCAST:
                    /* 'i' == class index */
                    i = buildClassIndex(i);
                    break;

                case GETFIELD:
                case PUTFIELD:
                    /* 'i' == field index */
```

```c
            /* i = buildFieldSignatureIndex(i); */
            i = buildStaticFieldSignatureIndex(i);
            break;

          case GETSTATIC:
          case PUTSTATIC:
            /* 'i' == field index */
            i = buildStaticFieldSignatureIndex(i);
            break;

          case INVOKEVIRTUAL:
          case INVOKENONVIRTUAL:
          case INVOKESTATIC:
          case INVOKEINTERFACE:
            /* 'i' == method signature index */
            i = buildSignatureIndex(i);
            break;
        }

        /* Insert application constant reference */
        code[pc+1] = (i >> 8) & 0xFF;
        code[pc+2] = i & 0xFF;
    }

#ifdef MODIFY_BYTECODE
    /* Translate codes */
    for (pc = 0; pc < codelen; pc = npc) {
        npc = nextPC(pc, code);
        code[pc] = codeinfo[code[pc]].translation;
    }
#endif

    /* Relink jumps */
    for (i = 0; i < jumpPos; i++) {
        apc = jumpTable[i].at;
        pc = jumpTable[i].from;
        npc = jumpTable[i].to - pc;

        code[apc+0] = (npc >> 8) & 0xFF;
        code[apc+1] = npc & 0xFF;
    }

    /* Fixup length */
    imeth->external->code_length = codelen;
    imeth->esize = (SIZEOFMETHOD + codelen + 3) & -4;
}
```
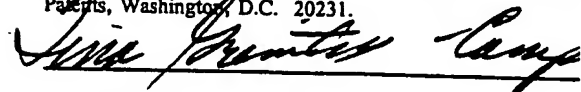
D

E

F

F

D-4

APPENDIX E

APPENDIX E

Example Loading And Execution Control Program

```
public class  Bootstrap (

    // Constants used throughout the program
    static final byte BUFFER_LENGTH           = 32;
    static final byte ACK_SIZE                 = (byte)1;
    static final byte ACK_CODE                 = (byte)0;
    static final byte OS_HEADER_SIZE           = (byte)0x10;
    static final byte GPOS_CREATE_FILE         = (byte)0xE0;

    static final byte ST_INVALID_CLASS         = (byte)0xC0;
    static final byte ST_INVALID_PARAMETER     = (byte)0xA0;
    static final byte ST_INS_NOT_SUPPORTED     = (byte)0xB0;
    static final byte ST_SUCCESS               = (byte)0x00;

    static final byte ISO_COMMAND_LENGTH       = (byte)5;
    static final byte ISO_READ_BINARY          = (byte)0xB0;
    static final byte ISO_UPDATE_BINARY        = (byte)0xD6;
    static final byte ISO_INIT_APPLICATION     = (byte)0xF2;
    static final byte ISO_VERIFY_KEY           = (byte)0x2A;
    static final byte ISO_SELECT_FILE          = (byte)0xA4;

    static final byte ISO_CLASS                = (byte)0xC0;
    static final byte ISO_APP_CLASS            = (byte)0xF0;


    public static void main () (

      byte pbuffer[] = new byte[ISO_COMMAND_LENGTH];
      byte dbuffer[] = new byte[BUFFER_LENGTH];
      byte ackByte[] = new byte[ACK_SIZE];
      //short fileId;
      short offset;
      byte bReturnStatus;

      // Initialize Communications
      _OS.SendATR();

      do (
          // Retrieve the command header
          _OS.GetMessage(pbuffer, ISO_COMMAND_LENGTH, ACK_CODE);

          // Verify class of the message - Only ISO + Application
          if ((pbuffer[0] != ISO_APP_CLASS)
           && (pbuffer[0] != ISO_CLASS)) (
              _OS.SendStatus(ST_INVALID_CLASS);
          )
          else (
            // go through the switch
            // Send the acknowledge code

            // Verify if data length too large
            if (pbuffer[4] > BUFFER_LENGTH) (
              bReturnStatus = ST_INVALID_PARAMETER;
            )
            else
            (
              switch (pbuffer[1]) (
              case ISO_SELECT_FILE:
                  // we always assume that length is 2
                  if (pbuffer[4] != 2) (
                      bReturnStatus = ST_INVALID_PARAMETER;
                  )
                  else
                  (
                      // get the fileId(offset) in the data buffer
                      _OS.GetMessage(dbuffer, (byte)2, pbuffer[1]);
                      // cast dbuffer[0..1] into a short
```

$E-1$

```
                                    offset = (short) ((dbuffer[0] << 8) | (dbuffer[1] & 0x00FF));
                                    bReturnStatus = _OS.SelectFile(offset);
                                }
                                break;

                        case ISO_VERIFY_KEY:
                            // Get the Key from the terminal
                            _OS.GetMessage(dbuffer, pbuffer[4], pbuffer[1]);

                            bReturnStatus = _OS.VerifyKey(pbuffer[3],
                                                          dbuffer,
                                                          pbuffer[4]);
                            break;

                        case ISO_INIT_APPLICATION:
                            // Should send the id of a valid program file
                            _OS.GetMessage(dbuffer, (byte)1, pbuffer[1]);
                            // compute fileId(offset) from pbuffer[2..3] via casting
                            offset = (short) ((pbuffer[2] << 8) | (pbuffer[3] & 0x00FF));
                            bReturnStatus = _OS.Execute(offset,
                                                        dbuffer[0]);
                            break;
                        case GPOS_CREATE_FILE:
                            if (pbuffer[4] != OS_HEADER_SIZE) {
                                bReturnStatus = ST_INVALID_PARAMETER;
                                break;
                            }
                            // Receive The data
                            _OS.GetMessage(dbuffer, pbuffer[4], pbuffer[1]);
                            bReturnStatus = _OS.CreateFile(dbuffer);
                            break;

                        case ISO_UPDATE_BINARY:
                            _OS.GetMessage(dbuffer, pbuffer[4], pbuffer[1]);
                            // compute offset from pbuffer[2..3] via casting
                            offset = (short) ((pbuffer[2] << 8) | (pbuffer[3] & 0x00FF));
                            // assumes that a file is already selected
                            bReturnStatus = _OS.WriteBinaryFile (offset,
                                                                 pbuffer[4],
                                                                 dbuffer);
                            break;
                        case ISO_READ_BINARY:
                            // compute offset from pbuffer[2..3] via casting
                            offset = (short) ((pbuffer[2] << 8) | (pbuffer[3] & 0x00FF));
                            // assumes that a file is already selected
                            bReturnStatus = _OS.ReadBinaryFile (offset,
                                                                pbuffer[4],
                                                                dbuffer);

                            // Send the data if successful
                            ackByte[0] = pbuffer[1];
                            if (bReturnStatus == ST_SUCCESS) {
                                _OS.SendMessage(ackByte, ACK_SIZE);
                                _OS.SendMessage(dbuffer, pbuffer[4]);
                            }
                            break;
                        default:
                            bReturnStatus = ST_INS_NOT_SUPPORTED;
                    }
                }
                _OS.SendStatus(bReturnStatus);
            }
        }
        while (true);
    }
}
```
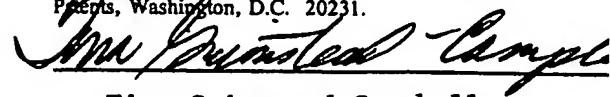
E·2

APPENDIX F

APPENDIX F

Methods For Accessing Card Operating System Capabilities
The Preferred Embodiment

```
public class _OS (

    static native byte      SelectFile          (short  file_id);
    static native byte      SelectParent        ();
    static native byte      SelectCD            ();
    static native byte      SelectRoot          ();

    static native byte      CreateFile          (byte   file_hdr[]);
    static native byte      DeleteFile          (short  file_id);

    // General File Manipulation
    static native byte      ResetFile           ();
    static native byte      ReadByte            (byte   offset);
    static native short     ReadWord            (byte   offset);

    // Header Manipulation
    static native byte      GetFileInfo         (byte   file_hdr[]);

    // Binary File support
    static native byte      ReadBinaryFile      (short  offset,
                                                 byte   data_length,
                                                 byte   buffer[]);
    static native byte      WriteBinaryFile     (short  offset,
                                                 byte   data_length,
                                                 byte   buffer[]);

    // Record File support
    static native byte      SelectRecord        (byte   record_nb,
                                                 byte   mode);
    static native byte      NextRecord          ();
    static native byte      PreviousRecord      ();


    static native byte      ReadRecord          (byte   record_data[],
                                                 byte   record_nb,
                                                 byte   offset,
                                                 byte   length);
    static native byte      WriteRecord         (byte   buffer[],
                                                 byte   record_nb,
                                                 byte   offset,
                                                 byte   length);


    // Cyclic File Support
    static native byte      LastUpdatedRec      ();

    // Messaging Functions
    static native byte      GetMessage          (byte   buffer[],
                                                 byte   expected_length,
                                                 byte   ack_code);
    static native byte      SendMessage         (byte   buffer[],
                                                 byte   data_length);
    static native byte      SetSpeed            (byte   speed);

    // Identity Management
    static native byte      CheckAccess         (byte   ac_action);
    static native byte      VerifyKey           (byte key_number,
                                                 byte key_buffer[],
                                                 byte key_length);
    static native byte      VerifyCHV           (byte   CHV_number,
                                                 byte   CHV_buffer[],
                                                 byte   unblock_flag);
    static native byte      ModifyCHV           (byte   CHV_number,
                                                 byte   old_CHV_buffer[],
                                                 byte   new_CHV_buffer[],
```

F-1

```
                                                          byte    unblock_flag);
        static native byte         GetFileStatus          ();
        static native byte         SetFileStatus          (byte    file_status);

        static native byte         GrantSupervisorMode ();
        static native byte         RevokeSupervisorMode();

        static native byte         SetFileACL             (byte    file_acl[]);
        static native byte         GetFileACL             (byte    file_acl[]);

        // File context manipulation
        static native void         InitFileStatus         ();
        static native void         BackupFileStatus       ();
        static native void         RestoreFileStatus      ();

        // Utilities
        static native byte         CompareBuffer          (byte    pattern_length,
                                                           byte    buffer_1[],
                                                           byte    buffer_2[]);
        static native short        AvailableMemory        ();
        static native void         ResetCard              (byte    mode);
        static native byte         SendATR                ();
        static native byte         SetDefaultATR          (byte    buffer[],
                                                           byte    length);
        static native byte         Execute    .           (short   file_id,
                                                           byte    flag);

        // Global state variable functions
        static native byte         GetIdentity            ();
        static native byte         GetRecordNb            ();
        static native short        GetApplicationId       ();
        static native byte         GetRecordLength        ();
        static native byte         GetFileType            ();
        static native short        GetFileLength          ();
        static native void         SendStatus             (byte status);

    )
```
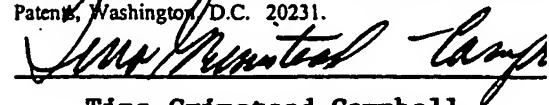
F-λ

APPENDIX G

Byte Code Attributes Tables

Dividing Java byte codes into type groups

Each bytecode is assigned a 5 bit type associated with it. This is used to group the codes into similarly t sets. In general this behaviour reflects how the types of byte codes operate on the stack, but types 0, 13, reflect specific kinds of instructions as denoted in the comments section.

The table below illustrates the state of the stack before and after each type of instruction is executed.

Type	Before execution	After exececution	Comment
0			Illegal instructi(
1	stk0==int stk1==int	pop(1)	
2	stk0==int	pop(1)	
3	stk0==int stk1==int	pop(2)	
4			
5	push(1)		
6	stk0==int stk1==int	pop(3)	
7	stk0==int	pop(1)	
8	stk0==ref	pop(1)	
9	stk0==int	pop(1)	
10	push(1)	stk0<-int	
11	push(1)	stk0<-ref	
12	stk0==ref	stk0<-int	
13			
14			DUPs, SWAP instruc
15			INVOKE instructior
16		stk0<-ref	FIELDS instructior

Using Standard Java Byte Code (without reordering) - Attribute Lookup Table

```
/*
 * Table of bytecode decode information.  This contains a bytecode type
 * and a bytecode length.  We currently support all standard bytecodes
 * (ie. no quicks) which gives us codes 0 to 201 (202 codes in all).
 */

#define T_      0
#define T3      1
#define T6      2
#define T1      3
#define T2      4
#define T7      5
#define T9      6
#define T8      7
#define T12     8
#define T10     9
#define T5      10
#define T11     11
#define T16     12
#define T4      13
#define T13     14
#define T14     15
#define T15     16

#define D(T,L)                              _BUILD_ITYPE_AND_ILENGTH(T, L)
#define _BUILD_ITYPE_AND_ILENGTH(T,L)       (_BUILD_ITYPE(T)|_BUILD_ILENGTH(L))
#define _BUILD_ITYPE(T)                     ((T) << 3)
#define _BUILD_ILENGTH(L)                   (L)
#define _GET_ITYPE(I)                       ((I) & 0xF8)
#define _GET_ILENGTH(I)                     ((I) & 0x07)

const uint8  _SCODE _decodeinfo[256] = {
        D( T4  , 1 ),           /* NOP           */
        D( T11 , 1 ),           /* ACONST_NULL   */
        D( T10 , 1 ),           /* ICONST_M1     */
        D( T10 , 1 ),           /* ICONST_0      */
        D( T10 , 1 ),           /* ICONST_1      */
        D( T10 , 1 ),           /* ICONST_2      */
        D( T10 , 1 ),           /* ICONST_3      */
        D( T10 , 1 ),           /* ICONST_4      */
        D( T10 , 1 ),           /* ICONST_5      */
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T10 , 2 ),           /* BIPUSH        */
        D( T10 , 3 ),           /* SIPUSH        */
        D( T_  , 2 ),           /* LDC1          */
        D( T11 , 3 ),           /* LDC2          */
        D( T_  , 3 ),
        D( T5  , 2 ),           /* ILOAD         */
        D( T_  , 2 ),
        D( T_  , 2 ),
        D( T_  , 2 ),
        D( T5  , 2 ),           /* ALOAD         */
        D( T5  , 1 ),           /* ILOAD_0       */
        D( T5  , 1 ),           /* ILOAD_1       */
        D( T5  , 1 ),           /* ILOAD_2       */
        D( T5  , 1 ),           /* ILOAD_3       */
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
```

```
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T5  , 1 ),          /* ALOAD_0     */
        D( T5  , 1 ),          /* ALOAD_1     */
        D( T5  , 1 ),          /* ALOAD_2     */
        D( T5  , 1 ),          /* ALOAD_3     */
        D( T_  , 1 ),          /* IALOAD      */
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),          /* AALOAD      */
        D( T7  , 1 ),          /* BALOAD      */
        D( T_  , 1 ),          /* CALOAD      */
        D( T7  , 1 ),          /* SALOAD      */
        D( T2  , 2 ),          /* ISTORE      */
        D( T_  , 2 ),
        D( T_  , 2 ),
        D( T_  , 2 ),
        D( T8  , 2 ),          /* ASTORE      */
        D( T2  , 1 ),          /* ISTORE_0    */
        D( T2  , 1 ),          /* ISTORE_1    */
        D( T2  , 1 ),          /* ISTORE_2    */
        D( T2  , 1 ),          /* ISTORE_3    */
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T8  , 1 ),          /* ASTORE_0    */
        D( T8  , 1 ),          /* ASTORE_1    */
        D( T8  , 1 ),          /* ASTORE_2    */
        D( T8  , 1 ),          /* ASTORE_3    */
        D( T_  , 1 ),          /* IASTORE     */
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),          /* AASTORE     */
        D( T6  , 1 ),          /* BASTORE     */
        D( T_  , 1 ),          /* CASTORE     */
        D( T6  , 1 ),          /* SASTORE     */
        D( T2  , 1 ),          /* POP         */
        D( T3  , 1 ),          /* POP2        */
        D( T13 , 1 ),          /* DUP         */
        D( T13 , 1 ),          /* DUP_X1      */
        D( T13 , 1 ),          /* DUP_X2      */
        D( T13 , 1 ),          /* DUP2        */
        D( T13 , 1 ),          /* DUP2_X1     */
        D( T13 , 1 ),          /* DUP2_X2     */
        D( T13 , 1 ),          /* SWAP        */
        D( T1  , 1 ),          /* IADD        */
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T1  , 1 ),
        D( T_  , 1 ),          /* ISUB        */
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T_  , 1 ),
        D( T1  , 1 ),          /* IMUL        */
        D( T_  , 1 ),
        D( T_  , 1 ),
```

```
                        D( T_  , 1 ),
                        D( T1  , 1 ),        /* IDIV           */
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T1  , 1 ),        /* IREM           */
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T9  , 1 ),        /* INEG           */
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T1  , 1 ),        /* ISHL           */
                        D( T_  , 1 ),
                        D( T1  , 1 ),        /* ISHR           */
                        D( T_  , 1 ),
                        D( T1  , 1 ),        /* IUSHR          */
                        D( T_  , 1 ),
                        D( T1  , 1 ),        /* IAND           */
                        D( T_  , 1 ),
                        D( T1  , 1 ),        /* IOR            */
                        D( T_  , 1 ),
                        D( T1  , 1 ),        /* IXOR           */
                        D( T_  , 1 ),
                        D( T4  , 3 ),        /* IINC           */
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T9  , 1 ),        /* INT2BYTE       */
                        D( T9  , 1 ),        /* INT2CHAR       */
                        D( T_  , 1 ),        /* INT2SHORT      */
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T2  , 3 ),        /* IFEQ           */
                        D( T2  , 3 ),        /* IFNE           */
                        D( T2  , 3 ),        /* IFLT           */
                        D( T2  , 3 ),        /* IFGE           */
                        D( T2  , 3 ),        /* IFGT           */
                        D( T2  , 3 ),        /* IFLT           */
                        D( T3  , 3 ),        /* IF_ICMPEQ      */
                        D( T3  , 3 ),        /* IF_ICMPNE      */
                        D( T3  , 3 ),        /* IF_ICMPLT      */
                        D( T3  , 3 ),        /* IF_ICMPGE      */
                        D( T3  , 3 ),        /* IF_ICMPGT      */
                        D( T3  , 3 ),        /* IF_ICMPLE      */
                        D( T3  , 3 ),        /* IF_ACMPEQ      */
                        D( T3  , 3 ),        /* IF_ACMPNE      */
                        D( T4  , 3 ),        /* GOTO           */
                        D( T_  , 3 ),        /* JSR            */
                        D( T_  , 2 ),        /* RET            */
                        D( T2  , 0 ),        /* TABLESWITCH    */
                        D( T2  , 0 ),        /* LOOKUPSWITCH   */
                        D( T2  , 1 ),        /* IRETURN        */
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T_  , 1 ),
                        D( T8  , 1 ),        /* ARETURN        */
                        D( T4  , 1 ),        /* RETURN         */
```
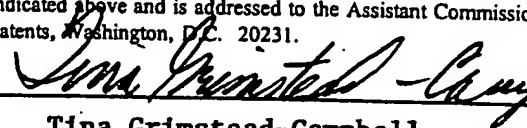
G-4

```
D( T15 , 3 ),          /* GETSTATIC      */
D( T15 , 3 ),          /* PUTSTATIC      */
D( T15 , 3 ),          /* GETFIELD       */
D( T15 , 3 ),          /* PUTFIELD       */
D( T14 , 3 ),          /* INVOKEVIRTUAL  */
D( T14 , 3 ),          /* INVOKESPECIAL  */
D( T14 , 3 ),          /* INVOKESTATIC   */
D( T14 , 5 ),          /* INVOKEINTERFACE */
D( T_  , 1 ),
D( T11 , 3 ),          /* NEW            */
D( T16 , 2 ),          /* NEWARRAY       */
D( T_  , 3 ),
D( T12 , 1 ),          /* ARRAYLENGTH    */
D( T8  , 1 ),          /* ATHROW         */
D( T16 , 3 ),          /* CHECKCAST      */
D( T12 , 3 ),          /* INSTANCEOF     */
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 4 ),
D( T8  , 3 ),          /* IFNULL         */
D( T8  , 3 ),          /* IFNONNULL      */
D( T_  , 5 ),
D( T_  , 5 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
D( T_  , 1 ),
```

```
D( T_   , 1 ),
D( T_   , 1 ),
D( T_   , 1 ),
D( T_   , 1 ),
D( T_   , 1 ),
D( T_   , 1 ),
D( T_   , 1 ),
);
```

G-6

APPENDIX H

APPENDIX H

Checks Done On Java Byte Codes By Type

Decoding the instruction. This gives us the length to generate the next PC, and the instruction type:

```
pcargl = _GET_ILENGTH(_decodeinfo[insn]);
itype = _GET_ITYPE(_decodeinfo[insn]);
```

Implement some pre-execution checks based on this:

```
/* Check the input stack state based on the instuction type */
if (itype <= ITYPE9) {
        if (itype <= ITYPE1) {
                check_stack_int(1);
        }
        check_stack_int(0);
}
else if (itype <= ITYPE12) {
        check_stack_ref(0);
}
else if (itype < ITYPE11) {
        push(1);
}
```

Finally, implement some post execution checks:

```
/* Set the output state */
if (itype <= ITYPE8) {
        if (itype <= ITYPE6) {
                if (itype >= ITYPE6) {
                        pop(1);
                }
                pop(1);
        }
        pop(1);
}
else if (itype <= ITYPE10) {
        set_stack_int(0);
}
else if (itype >= ITYPE11 && itype <= ITYPE16) {
        set_stack_ref(0);
}
```
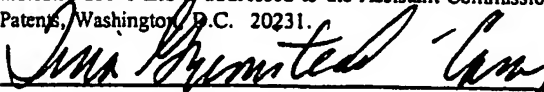
APPENDIX I

APPENDIX I

Checks Done On Renumbered Java Byte Codes

Get the instruction. The numeric value of the instruction implicitly contains the instruction type:

```
insn = getpc(-1);
```

Implement some pre-execution checks based on this:

```
/*
 * Check input stack state. By renumbering the byte codes we can
 * perform the necessary security checks by testing if the value of the
 * byte code (and hence the byte code) belongs to the correct group
 */
if (insn <= TYPE9_END) {
  if (insn <= TYPE1_END) {
    check_stack_int(1);
  }
  check_stack_int(0);
}
else if (insn <= TYPE12_END) {
  check_stack_ref(0);
}
else if (insn <= TYPE11_END) {
  push(1)
}
```

Finally, implement some post execution checks:

```
/*
 * Set output stack state.
 */
if (insn <= TYPE8_END) {
  if (insn <= TYPE6_END) {
    if (insn >= TYPE6_START) {
      pop(1);
    }
    pop(1);
  }
  pop(1);
}
else if (insn <= TYPE10_END) {
  set_stack_int(0);
}
else if (insn >= TYPE11_START && insn <= TYPE16_END) {
  set_stack_ref(0);
}
```

I-1

Reordering of supported Java byte codes by type

```
/* TYPE 3 */

#define s_POP2              0
#define s_IF_ICMPEQ         1
#define s_IF_ICMPNE         2
#define s_IF_ICMPLT         3
#define s_IF_ICMPGE         4
#define s_IF_ICMPGT         5
#define s_IF_ICMPLE         6
#define s_IF_ACMPEQ         7
#define s_IF_ACMPNE         8

/* TYPE 6 */

#define TYPE6_START         9

#define s_SASTORE           9
#define s_AASTORE           10
#define s_BASTORE           11

#define TYPE6_END           12

/* TYPE 1 */

#define s_IADD              13
#define s_ISUB              14
#define s_IMUL              15
#define s_IDIV              16
#define s_IREM              17
#define s_ISHL              18
#define s_ISHR              19
#define s_IUSHR             20
#define s_IAND              21
#define s_IOR               22
#define s_IXOR              23

#define TYPE1_END           23

/* TYPE 2 */

#define s_ISTORE            24
#define s_POP               25
#define s_IFEQ              26
#define s_IFNE              27
#define s_IFLT              28
#define s_IFGE              29
#define s_IFGT              30
#define s_IFLE              31
#define s_TABLESWITCH       32
#define s_LOOKUPSWITCH      33
#define s_IRETURN           34

/* TYPE 7 */

#define s_SALOAD            35
#define s_AALOAD            36
#define s_BALOAD            37

/* TYPE 9 */

#define s_INEG              39
#define s_INT2BYTE          40
#define s_INT2CHAR          41

#define TYPE9_END           41

/* TYPE 8 */

#define s_ASTORE            42
#define s_ARETURN           43
```

I-2

```c
#define s_ATHROW            44
#define s_IFNULL            45
#define s_IFNONNULL         46
                            46

/* TYPE 12 */

#define s_ARRAYLENGTH       47
#define s_INSTANCEOF        48

#define TYPE12_END          48

/* TYPE 10 */

#define s_SIPUSH            49

#define TYPE10_END          49

/* TYPE 5 */

#define s_ILOAD             50
#define s_ALOAD             51

/* TYPE 11 */

#define TYPE11_START        52

#define s_ACONST_NULL       52
#define s_LDC2              53
#define s_JSR               54
#define s_NEW               55

#define TYPE11_END          55

/* TYPE 16 */

#define s_NEWARRAY          56
#define s_CHECKCAST         57

#define TYPE16_END          57

/* TYPE 13 */

#define s_DUP               58
#define s_DUP_X1            59
#define s_DUP_X2            60
#define s_DUP2              61
#define s_DUP2_X1           62
#define s_DUP2_X2           63
#define s_SWAP              64

/* TYPE 14 */

#define s_INVOKEVIRTUAL     65    /* 01000001 */
#define s_INVOKENONVIRTUAL  66    /* 01000010 */
#define s_INVOKESTATIC      67    /* 01000011 */
#define s_INVOKEINTERFACE   68    /* 01000100 */

/* TYPE 15 */

#define s_GETSTATIC         69
#define s_PUTSTATIC         70
#define s_GETFIELD          71
#define s_PUTFIELD          72

/* TYPE 4 */

#define s_NOP               73
#define s_IINC              74
#define s_GOTO              75
#define s_RET               76
#define s_RETURN            77
```

I-3